Collaborative Merge in Distributed Software Development: Who Should Participate?

Catarina Costa[1, 3]
[1]Technology and Exacts Science Center
Federal University of Acre, UFAC
Rio Branco - AC, Brazil
catarina@ufac.br

José J. C. Figueirêdo[2]
[2]Information Technology Department
Federal Institute of Acre, IFAC
Rio Branco – AC, Brazil
jair.figueiredo@ifac.edu.br

Leonardo Murta[3]
[3]Computing Institute
Fluminense Federal University, UFF
Niterói - RJ, Brazil
leomurta@ic.uff.br

Abstract— **Merge conflicts, which are rather common throughout the process of software development, are more frequent and complex to resolve when using the distributed software development approach, where developers are geographically dispersed. Normally, in the case of workspace merge, the last developer to merge code is responsible of conciliating the changes made in parallel and resolving conflicts. However, the last developer is not always the best team member to complete this task because he or she may not be familiar with the other parts of the code. With this in mind, the goal of this work is to analyze merge profiles of eight software projects and check if the development log is an appropriate source of information for identifying the key participants for collaborative merge. The obtained results are promising in this direction.**

Keywords-** **Collaborative merge; distributed software development; version control

I. Introduction

According to Bendix et al. [1], Configuration Management can assist either co-located or distributed software development. Besides the existence of similarities between both approaches, there are some important differences. One of them is how each approach handles conflicts. When a conflict arises in a co-located project, developers can gather together and work over it. Distributed projects, which aside from the geographic dispersion of the team may involve time and sociocultural differences, require a deeper effort to bypass this kind of impasse.

Merge conflicts are common, especially in massive and distributed software projects [2]. Identifying and resolving these conflicts are not trivial tasks. While two simple concurrently changes might be individually correct, their combination may raise conflicts. These conflicts demand conciliating choices and, consequently, this whole scenario usually introduces rework [3].

Most contemporary version control systems adopt optimistic version control and are based on the premise that the workspace merge is done by the last developer to commit. For this reason, the developer's knowledge regarding the changes performed in parallel is not taken into consideration. Consequently, the last developer might not be the most suitable developer to fulfill the conciliation of decisions during merge.

Some state-of-the-practice tools, such as KDiff [1], WinMerge[2], Diffuse[3], and SourceGear DiffMerge[4], feature automated solutions for merge conflicts, but they not able to resolve conflicts that occur in the same region of the files. In contrast, state-of-the-practice tools, such as CASI [4], CloudStudio [5], CoDesign [6], Palantír [3], and Syde [7], try to avoid conflicts through awareness. Their main goal is to alert developers in a proactive way of situations that may lead to conflicts [5], [8]. However, these approaches require certain conditions, such as developers working in sync in terms of time and over the same branch, which is not always the case for distributed software development. All in all, even adopting the aforementioned tools, conflicts may still exist. Finding the appropriate developers to solve them is an important challenge.

The use of the existing shared knowledge of software projects is a missing element for conflict resolution in distributed software development [2]. The configuration management repositories have a trail of who changed the software and can help on identifying the most appropriate developers for solving conflicts. This information could be extracted, for instance, from the project development progress history until the point where the merge takes place.

This work aims at providing a deeper analysis of how parallel development occurs and who could participate in collaborative merge sessions. To do so, we analyzed the merge profiles of eight software projects versioned using the Git[5]. The analysis was guided by the following five research questions, whose objective is the identification of the developers that worked in parallel before a merge and that should take part in the merge process.

Q1: What is the average number of developers per branch?

[1] http://kdiff3.sourceforge.net/
[2] http://winmerge.org/
[3] http://diffuse.sourceforge.net/
[4] http://www.sourcegear.com/diffmerge/
[5] http://git-scm.com

Q2: What is the average number of commits per developer in each project?

Q3: What is the maximum number of developers in each branch?

Q4: How many merges have the same developers in both branches?

Q5: Who should participate in collaborative merge sessions?

These answers can provide important insights on how much the project development log can assist in the designation of key participants to resolve merge conflicts. This information can help on defining new methods for assign developers to participate in merge sessions, known as collaborative merge.

This work is organized in six sections, including this introductory section. Section II presents the concepts involved in the research. Section III features the related works. Section IV presents the research method. Section V contains the analysis results and some insights on how the development log could help on the identification of merge participants. Finally, the conclusions of this study are presented in Section VI.

II. Background

According to Mens [9], three types of conflicts may arise during merge: textual, syntactic, and semantic. Textual conflicts, also known as physical conflicts, occur when concurrent operations (e.g., addition, removal, or edition) take place over the same physical parts of a file (e.g., same line). Syntactic conflicts occur when concurrent operations break the syntactic structure of the file when combined. The syntactic structure of a file is usually defined in terms of a schema or grammar. Finally, semantic conflicts occur when concurrent operations break the semantics of the file when combined. The semantics of the file can be expressed both in terms of the programming language semantics or expected program behavior (usually verified by test cases).

The existing version control systems are limited to textual conflicts. This way, a conflict arises when two or more developers make inconsistent changes in the same part of the code. When this happens, the changes cannot be incorporated into the repository until the conflict is resolved by some developer [10]. Fig. 1 shows an example of merge, where four developers work in two branches, being d1, d2, and d3 in the first branch, and d2, d1, and d4 in the second branch. In a given moment, d4 decides to merge the second branch into the first branch. In this case, in most version control systems, d4 must conciliate the choices and possibly edit some parts of the code written by the other developers.

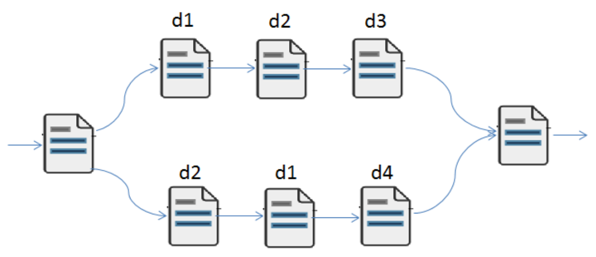

Figure 1. A software merge scenario.

This example evidences that, disregarding the content contribution of each developer, d1 and d2 probably know the part they are working with better than d3 and d4 for that merge. This occurs because they worked in both branches and, consequently, understand better the changes performed in both branches. The developer d4 could be at least assisted by d1 and/or d2 to more effectively conciliate the decisions.

III. Related Work

The majority of the works in the literature on merge conflicts focus on the prevention of conflicts through awareness. By monitoring workspaces, awareness tools, such as Palantír [3] and CloudStudio [5], notify the developers every time an change that may lead to a conflict is being made. Other approaches, such as Crystal [10] and WeCode [11], work by continuously trying to merge the local copies of the various users. Although the works might have distinct solutions, conflict prevention is the goal of these approaches.

All approaches presented here require developers to be constantly synced in time throughout the development of the software, but this is not always possible in distributed software development. As mentioned before, the related work also expects that developers are working over the same branch. However, for different reasons [12] sub-teams work in parallel over different branches. Thus, conflicts would not be preemptively detected, which would imply solving them during merge anyway.

Nieminen's work [2] proposes a web-based tool (CoRED) that allows multiple users to resolve merge conflicts in a collaborative way. When facing a conflict, developers can invite a workmate to take part in a conflict resolution session, aiming at reaching a consensus. In this work, the authors focus on conflict resolution with the assistance of a web tool, and the developer select the resolution session participant of their choice. Although relevant, this work can be seen as complementary to our work, as they do not mention how participants could be selected.

In other work, based in collaborative merge of models [13], the authors present a model merging approach that shows conflicts as open questions. To do so, the approach allows to represent conflicts explicitly as parts of the model and thereby to postpone them. This facilitates discussion on conflicts and collaborative conflict resolution. The study shows that developers like to discuss conflicts in a considerable percentage of conflicts. The authors defend that the resolution of a conflict must not be assigned to only one developer, becoming a collaborative task.

The last two works discussed in this section contributed with interesting insights for this paper. Our work differs from them because we aim at analyzing merge profiles and check if the development log is an appropriate source of information for identifying the key participants for collaborative merge. Besides, these works do not foresee the physical distance of the people involved in the project, which makes it harder to identify the developer that has to be contacted when a conflict arises.

IV. Materials and Methods

Our research followed the process depicted in Fig. 2 for comprehending how merge happens in practice. This process comprises project selection, merge profile analysis, and consolidation of results.

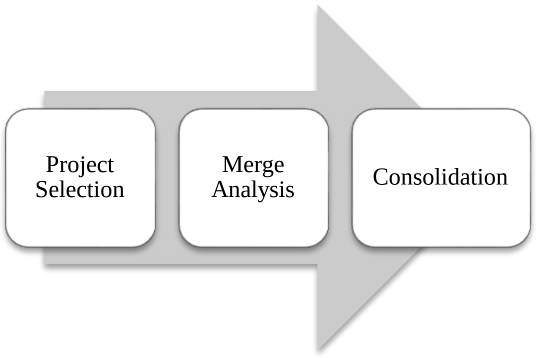

Figure 2. Research Steps.

In the **project selection step**, eight projects were chosen: Django, Git, JQuery, MaNGOS, Perl5, Rails, Voldemort, and Zotero. These projects were selected because they have different sizes, are popular, are coded in different programming languages, have different goals, and are very active. This contributes for a more embracing analysis. After select these projects, they were cloned from Github[6] making it possible to perform the next step of our research process.

In the **Merge Analysis Step**, the log of each selected project was analyzed to extract information about developers, commits, and merges. To do so, we implemented an automated infrastructure that parses the log provided by Git and extracts all merge commits. Moreover, it is able to identify the parent commits that were merged and navigate until the common ancestor just before forking the history.

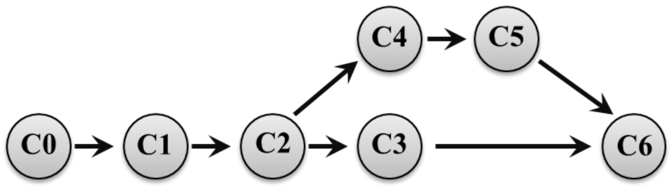

Figure 3. A simple merge example

Fig. 3 shows an example of merge commit (C6), two parent commits that were merged (C3 and C5), and the common ancestor (C2). From these commits, it is possible to identify the commits within each branch. These commits are comprised between the common ancestor and each one of the parent commits that were merged (including the parent commits). The first branch has two commits (C4 and C5) and the second branch has only one commit (C3).

After identifying each commit that was merged, we extract the author information (name and e-mail). Next, we extract multiple information, such as: who committed in each branch, how many commits they did, who committed in both branches, and others.

The most complex merges were prioritized by inverse sorting the merges by the harmonic mean of the number of developers in each branch. An in-depth analysis was carried out for some of the most complex merge scenarios of these projects in order to investigate and indicate the developer to participate in the collaborative merge.

Finally, the **consolidation step** comprises the answers of the questions presented in Section I. The results were organized in a way that shows the merge profiles and information on the most suitable developers to take part in conflict resolution, answering the research questions.

[6] https://github.com/

V. Results and Discussion

This section presents the answers to the research questions posed on Section I, according to the research process described in Section IV. Besides getting merge behavior patterns, information such as the average number of commits by developer and the number of developers working in branches, amongst other, are shown. Table I describes the analyzed projects with their total number of merges and developers. This data was extracted from projects between the end of 2013 and beginning of 2014.

TABLE I. Project's General Description

Project	**General Information**		
	Purpose	***# Merges***	***# Developers***
Django	Python web framework	404	396
Git	Distributed revision control system	7434	1103
JQuery	JavaScript Library	250	133
MaNGOS	A full-featured MMO server suite	255	266
Perl5	Programming language	1607	789
Rails	Ruby web framework	6147	2528
Voldemort	Distributed database	406	59
Zotero	Citation manager	223	16

By analyzing the information shown on Table I, it is noticeable that the projects range from just few merges (Zotero, JQuery, and MaNGOS) to a large number of merges (Git and Rails). It is also observable that the projects have varied numbers of participants, ranging from 16 in Zotero to 2528 in the Rails project. This diversity is important to allow us exploring different situations, enabling a wider observation over the development projects.

In the following, we answer each research question presented in Section I.

A. *Q1: What is the average number of developers per branch?*

The first question observes the quantity of developers involved in each branch of the merges, for each project.

The branches of all the merges, which comprise commits between each parent of the merge and the common ancestor, were processed and the results are shown in Table II. It is important to notice that there is a reasoning to number the branches: "the first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in" [14].

TABLE II. AVERAGE NUMBER OF DEVELOPERS PER BRANCH

Project	**Developers per Branch**			
	Average Branch 1	***Standard Deviation***	***Average Branch 2***	***Standard Deviation***
Django	7.05	20.60	2.64	8.88
Git	24.00	53.63	1.99	14.01
JQuery	4.56	7.26	1.57	2.07
MaNGOS	1.58	1.86	5.00	4.88
Perl5	2.02	7.01	3.46	6.51
Rails	10.95	43.77	2.41	8.58
Voldemort	1.47	1.67	1.73	1.45
Zotero	1.38	1.16	1.56	0.94

It is observable that, for example, Git has an average of 24 developers per merge on branch 1 and only 1.99 developers on branch 2. The same occurs in other three projects, having high values on branch 1: Django, JQuery, and Rails. In contrast, MaNGOS, Perl5, Voldemort, and Zotero have a higher average of developers on branch 2. However, with a clear exception of MaNGOS, the difference is not significant.

B. Q2: What is the average number of commits per developer in each project?

The second question observes the average number of commits per developer. To get the numbers in Table III, the totality of commits was divided by the totality of project participants.

TABLE III. AVERAGE COMMITS PER DEVELOPER

Projects Name	**Commits per Developer**	
	Average	***Standard Deviation***
Django	10.88	17.49
Git	176.26	418.56
JQuery	12.52	24.08
MaNGOS	7.31	21.37
Perl5	12.17	50.25
Rails	33.51	100.31
Voldemort	23.08	43.12
Zotero	42.06	74.08

The average number of commits per developer evidences the average participation of the developers in the project. Furthermore, it allows for the identification of the developers that participate the most and, therefore, being the most suitable to resolve merge conflicts.

C. Q3: What is the maximum number of developers in each branch?

The third research question observes the maximum number of developers in each of the merging branches. Table IV shows the merges with the highest harmonic mean of branches 1 and 2. The harmonic mean is a plausible metric because it allows for finding scenarios with high values in both branches.

TABLE IV. MAXIMUM NUMBER OF DEVELOPERS PER BRANCH

Projects Name	**Maximum Number of Developers**			
	Merge Commit Hash	***Branch 1***	***Branch 2***	***Harmonic Mean***
Django	9cc6cfc4057e…	8	12	9.60
Git	a8816e7bab03…	158	10	18.80
JQuery	63aaff590ccc…	20	3	5.22
MaNGOS	404785091845…	11	17	13.35
Perl5	fed34a19f844…	10	6	7.50
Rails	bf2b9d2de3f8…	47	164	49.37
Voldemort	2a5c69145fb6…	4	8	5.33
Zotero	d456117ebe9c	3	5	3.75

Consider, for example, the merge with most developers in the Rails project, with harmonic mean of 49.37, having 47 developers that performed commits on branch 1 and 164 developers that performed commits on branch 2. For this case, the complexity of identifying the key participant(s) to solve a certain merge conflict can be rather high. The possibilities of communication exceed 7,000 channels between the involved developers. On the other hand, the Zotero project has harmonic mean of 3.75, leading to 15 different channels of communication amongst developers.

The amount of developers that performed commits in a given branch can help identify which participant is more suitable to participate in the collaborative merge. That is, who is a potential knower of the changes made in such merge.

D. Q4: How many merges have the same developers in both branches?

The fourth question observes the number of merges having the same developers in both branches. Table V displays information on the intersections of developers in merging branches. The first column presents the project names. The remaining columns show the quantity and proportion of:

a) Merges that have all the developers in common in the branches;

b) Merges that have some developers in common; and

c) Merges that have different developers in both branches.

As seen in the Django project, only 1 merge (0.25%) has exactly the same developers in both branches and 8.66% have some intersection. In the Git project, for instance, the intersection reaches more than 50% of the merges, and the same developers participate in both branches in about 5% of the merges. In the JQuery, MaNGOS, Voldemort, and Zotero projects, the number of people in both branches is also expressive, with more than 30% of the merges having some intersection. The Django, Perl, and Rails projects feature small intersections of developers in the branches.

TABLE V. INTERSECTION OF DEVELOPERS IN MERGING BRANCHES

Projects Name	**Intersection of Developers**					
	Exactly the same (a)		***Some intersection (b)***		***Without intersections (c)***	
Django	1	0.25%	35	8.66%	368	91.09%
Git	435	5.85%	3926	52.82%	3073	41.33%
JQuery	19	7.60%	72	28.80%	159	63.60%
MaNGOS	7	2.74%	71	27.85%	177	69.41%
Perl5	15	0.93%	95	5.91%	1497	93.16%
Rails	56	0.91%	886	14.41%	5205	84.68%
Voldemort	64	15.76%	83	20.44%	259	63.80%
Zotero	34	15.25%	94	42.15%	95	42.60%

Given the merge example presented on Fig. 1 on Section II, the information that there are developers that know each of the merging branches can aid in the designation of key participants to take part in the collaborative merge. It is believed that, as the developer participating on the merge knows parts of the work that is being completed in both branches, the chances of performing a more coherent choice during conciliation increases.

E. Q5: Who should participate in collaborative merge sessions?

In the fifth and last question we analyze some complex merge scenarios enlisted in Table IV and propose the most suitable participants to perform collaborative merge, should a conflict arise.

1) Git: the merge with the most complex setting of developers, by harmonic mean, has 158 developers in branch 1, and 10 developers in branch 2. Besides, 5 developers committed in both branches. Table VI shows the amount of commits made by each one of these 5 developers.

TABLE VI. DEVELOPERS IN BOTH BRANCHES FOR GIT

Merge	**Developers in Both Branches – Git Project**		
	Developers	***Commits in Branch 1***	***Commits in Branch 2***
a8816e7bab03 …	Markus	23	5
	Alex	7	1
	Shawn	2	4
	Jens	1	4
	Ferry	1	1

For this merge, the average number of commits per developer is 8.9 in branch 1 and 2.3 in branch 2. In this case, although *Markus* was not the last developer to commit, should a conflict arise, he would be a strong candidate to participate in the collaborative merge.

In a collaborative merge based on logs, aside from the verification of the developer in both branches (Table VI), it is possible to verify the participants that performed most commits (Table VII). Table VII shows the six developers that committed the most, three in the first branch and three in the second branch. Besides possibly knowing a lot about the work being completed, if there is an intersection between the two analyses, the weight of the indication of the participant must be higher. For this merge, there are some intersections between developers in both analyses. Therefore, *Junio*, *Markus*, *Shawn*, and *Jens* would be candidates to be part of the collaborative merge.

TABLE VII. DEVELOPERS WITH MOST COMMITS

Merge	**Developers with the Most Commits – Git Project**		
	Developers	***Commits in Branch 1***	***Commits in Branch 2***
a8816e7bab03 …	Junio	**512**	0
	Schindelin	**69**	0
	Jeff	**58**	0
	Markus	23	**5**
	Shawn	2	**4**
	Jens	1	**4**

2) Perl5: the merge with the most complex scenario of developers, by harmonic mean, has 10 developers in branch 1 and 6 developers in branch 2. Only 1 developer participated in both branches, as seen on Table VIII.

TABLE VIII. DEVELOPERS IN BOTH BRANCHES FOR PERL5

Merge	**Developers in Both Branches – Perl5 Project**		
	Developers	***Commits in Branch 1***	***Commits in Branch 2***
fed34a19f844…	Hans	2	1

For this merge, only *Hans* has committed in both branches and so he is a strong candidate to participate in the collaborative merge. In addition, he was also one of the six developers that committed the most (Table IX).

TABLE IX. DEVELOPERS WITH MOST COMMITS

Merge	**Developers with the Most Commits – Perl5 Project**		
	Developers	**Commits in Branch 1**	**Commits in Branch 2**
fed34a19f844…	Gurusamy	**17**	0
	Hugo	**2**	0
	Tom	**2**	0
	Jarkko	0	**16**
	Hans	2	**1**
	Kurt	0	**1**

Table IX shows the six developers that committed the most, three in the first branch and three in the second branch. Two other strong candidates to participate in the merge are *Gurusamy* and *Jarkko*, who were responsible for a large number of commits on branches 1 and 2, respectively, as shown on Table IX.

3) Voldemort: the merge with the most complex setting of developers, by harmonic mean, has 4 developers in branch 1 and 8 developers in branch 2. In this case, 4 developers participated in both branches, as shown in Table X.

TABLE X. DEVELOPERS IN BOTH BRANCHES FOR VOLDEMORT

Merge	**Developers with the Most Commits – Voldemort Project**		
	Developers	***Commits in Branch 1***	***Commits in Branch 2***
2a5c69145fb6 …	Bhusesh	33	11
	Kirktrue	29	10
	Alex	20	13
	Bbansal	4	2

The developers who committed the most on branch 1 also committed on branch 2, which makes all of them (especially *Bhusesh*, *Kirktrue*, and *Alex*) strong candidates to perform the collaborative merge. In this case, the developers that participate in both branches were also the ones that performed the most commits, reinforcing the evidence that they should participate in the collaborative merge.

VI. CONCLUSION

As shown in the previous sections, it is possible to use the history log to identify the potentially best developers to take part in the collaborative merge session. This can be done both through the identification of developers who committed in both branches or the ones that committed the most in one of the branches.

As a contribution, this study presented answers to some research questions regarding repository logs. Besides, some merge analyses were displayed, showing that it is possible to identify potential developers to perform collaborative merges.

As a threat to validity, the analyses were based on the committers' names and/or email addresses, which can generate inconsistency in the numbers presented. Furthermore, the analyses were carried out in merges that had already occurred. For this reason, it was impossible to effectively measure the effects of including the suggested developers in the merge session. Moreover, this study does not take under consideration developers' previous knowledge of libraries or programming languages being used, but solely the commit logs, that is, the participation of the developers in the specific project under analysis.

As a future work, we hope to apply this research in pending merges, in order to observe the advantages of indicating who is the most suitable developer. Moreover, we intend to conceive a method for indicating the most suitable developers and automated such method. Finally, we want to add additional analysis related to the expertise over changed artifacts and the knowledge proximity amongst developers.

ACKNOWLEDGMENT

The authors would like to thank CAPES, CNPq, and FAPERJ for the financial support.

REFERENCES

[1] L. Bendix, J. Magnusson, and C. Pendleton, "Configuration Management Stories from the Distributed Software Development Trenches," presented at the 2012 IEEE Seventh International Conference on Global Software Engineering (ICGSE), 2012, pp. 51 –55.

[2] A. Nieminen, "Real-time collaborative resolving of merge conflicts," in 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012, pp. 540–543.

[3] A. Sarma, D. Redmiles, and A. van der Hoek, "Palant&#237;r: Early Detection of Development Conflicts Arising from Parallel Code Changes," IEEE Trans. Softw. Eng., vol. 38, no. 4, pp. 889–908, Jul. 2012.

[4] J. Portillo-Rodriguez, A. Vizcaino, C. Ebert, and M. Piattini, "Tools to Support Global Software Development Processes: A Survey," in 2010 5th IEEE International Conference on Global Software Engineering (ICGSE), 2010, pp. 13–22.

[5] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Unifying Configuration Management with Merge Conflict Detection and Awareness Systems," in Proceedings of the 2013 22nd Australian Conference on Software Engineering, Washington, DC, USA, 2013, pp. 201–210.

[6] J. young Bang, D. Popescu, G. Edwards, N. Medvidovic, N. Kulkarni, G. M. Rama, and S. Padmanabhuni, "CoDesign: a highly extensible collaborative software modeling framework," presented at the 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, vol. 2, pp. 243 –246.

[7] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," presented at the 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, vol. 2, pp. 235 –238.

[8] C. Costa and L. Murta, "Version Control in Distributed Software Development: A Systematic Mapping Study," in 2013 IEEE 8th International Conference on Global Software Engineering (ICGSE), 2013, pp. 90–99.

[9] T. Mens, "A state-of-the-art survey on software merging," Software Engineering, IEEE Transactions on, vol. 28, no. 5, pp. 449–462, 2002.

[10] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 168–178.

[11] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in Proceedings - International Conference on Software Engineering, 2012, pp. 342–352.

[12] S. P. Berczuk and B. Appleton, Software configuration management patterns: effective teamwork and practical integration. Boston, Mass: Addison-Wesley, 2003.

[13] M. Koegel, H. Naughton, J. Helming, and M. Herrmannsdoerfer, "Collaborative model merging," in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, New York, NY, USA, 2010, pp. 27–34.

[14] S. Chacon, Pro Git. Berkeley, CA; New York: Apress ; Distributed to the Book trade worldwide by Springer-Verlag, 2009.